Introducción

Este documento proporciona convenciones de codificación para el código Python que comprende la biblioteca estándar en la distribución principal de Python. Consulte el PEP informativo adjunto que describe las pautas de estilo para el código C en la implementación C de Python [1] .

Este documento y PEP 257 (Convenciones de cadenas de documentos) fueron adaptados del ensayo original de la Guía de estilo Python de Guido, con algunas adiciones de la guía de estilo de Barry [2] .

Esta guía de estilo evoluciona con el tiempo a medida que se identifican convenciones adicionales y las convenciones pasadas se vuelven obsoletas debido a cambios en el propio lenguaje.

Muchos proyectos tienen sus propias pautas de estilo de codificación. En caso de conflicto, dichas guías específicas del proyecto tienen prioridad para ese proyecto.

Una consistencia tonta es el duende de las pequeñas mentes

Una de las ideas clave de Guido es que el código se lee con mucha más frecuencia de lo que se escribe. Las pautas proporcionadas aquí están destinadas a mejorar la legibilidad del código y hacerlo coherente en todo el amplio espectro del código Python. Como dice PEP 20 , "La legibilidad cuenta".

Una guía de estilo tiene que ver con la coherencia. La coherencia con esta guía de estilo es importante. La coherencia dentro de un proyecto es más importante. La coherencia dentro de un módulo o función es lo más importante.

Sin embargo, debes saber cuándo ser inconsistente: a veces, las recomendaciones de la guía de estilo simplemente no son aplicables. En caso de duda, utilice su mejor criterio. Mire otros ejemplos y decida qué se ve mejor. ¡Y no dudes en preguntar!

En particular: ¡no rompa la compatibilidad con versiones anteriores solo para cumplir con este PEP!

Algunas otras buenas razones para ignorar una pauta en particular:

1. Al aplicar la directriz, el código sería menos legible, incluso para alguien que esté acostumbrado a leer el código que sigue este PEP.
2. Para ser coherente con el código circundante que también lo rompe (quizás por razones históricas), aunque esta también es una oportunidad para limpiar el desorden de otra persona (en el verdadero estilo XP).

3. Porque el código en cuestión es anterior a la introducción de la directriz y no hay otra razón para modificar ese código.
4. Cuando el código debe seguir siendo compatible con versiones anteriores de Python que no son compatibles con la función recomendada por la guía de estilo.

Diseño de código

Sangría

Utilice 4 espacios por nivel de sangría.

Las líneas de continuación deben alinear los elementos envueltos ya sea verticalmente usando la línea implícita de Python que se une entre paréntesis, corchetes y llaves, o usando una *sangría francesa* [7] . Cuando se utiliza una sangría francesa, se debe considerar lo siguiente; no debe haber argumentos en la primera línea y se debe usar más sangría para distinguirse claramente como una línea de continuación:

```
# Correcto:



# Alineado con el delimitador de apertura.

foo = nombre_función_larga (var_one, var_two,

                          var_three, var_four)



# Agregue 4 espacios (un nivel adicional de sangría) para distinguir los
argumentos del resto.

def nombre_función_larga (

        var_one, var_two, var_three,

        var_four):

    imprimir (var_one)



# Las sangrías colgantes deben agregar un nivel.

foo = nombre_función_larga (

    var_one, var_two,
```

```
    var_three, var_four)

# Equivocado:



# Se prohíben los argumentos en la primera línea cuando no se utiliza la
alineación vertical.

foo = nombre_función_larga (var_one, var_two,

    var_three, var_four)



# Se requiere más sangría ya que la sangría no es distinguible.

def nombre_función_larga (

    var_one, var_two, var_three,

    var_four):

    imprimir (var_one)
```

La regla de los 4 espacios es opcional para las líneas de continuación.

Opcional:

```
# Las sangrías colgantes * pueden * tener sangría diferente a 4 espacios.

foo = nombre_función_larga (

  var_one, var_two,

  var_three, var_four)
```

Cuando la parte condicional de una declaración `if` es lo suficientemente larga como para requerir que se escriba en varias líneas, vale la pena señalar que la combinación de una palabra clave de dos caracteres (es decir, `if`), más un espacio único, más un paréntesis de apertura crea un natural Sangría de 4 espacios para las líneas subsiguientes del condicional multilínea. Esto puede producir un conflicto visual con el conjunto de código con sangría anidado dentro de la declaración `if` , que naturalmente también estaría sangrado en 4 espacios. Este PEP no toma una posición explícita sobre cómo (o si) distinguir visualmente más tales líneas condicionales del conjunto anidado dentro de la declaración `if` . Las opciones aceptables en esta situación incluyen, pero no se limitan a:

```
# Sin sangría adicional.

si (this_is_one_thing y

    eso_es_otra_cosa):

    hacer algo()



# Agregue un comentario, que proporcionará cierta distinción en los editores

# Resaltado de sintaxis compatible.

si (this_is_one_thing y

    eso_es_otra_cosa):

    # Dado que ambas condiciones son verdaderas, podemos frobnicar.

    hacer algo()



# Agregue un poco de sangría adicional en la línea de continuación
condicional.

si (this_is_one_thing

        y eso_es_otra_cosa):

    hacer algo()
```

(Consulte también la discusión sobre si romper antes o después de los operadores binarios a continuación).

La llave / corchete / paréntesis de cierre en construcciones multilínea puede alinearse bajo el primer carácter que no sea un espacio en blanco de la última línea de la lista, como en:

```
my_list = [

    1, 2, 3,

    4, 5, 6,

    ]

result = some_function_that_takes_arguments (
```

```
    'a B C',

    'd', 'e', 'f',

    )
```

o puede estar alineado bajo el primer carácter de la línea que inicia la construcción multilínea, como en:

```
my_list = [

    1, 2, 3,

    4, 5, 6,

]

result = some_function_that_takes_arguments (

    'a B C',

    'd', 'e', 'f',

)
```

¿Pestañas o espacios?

Los espacios son el método de sangría preferido.

Las pestañas deben usarse únicamente para mantener la coherencia con el código que ya está sangrado con pestañas.

Python 3 no permite mezclar el uso de pestañas y espacios para la sangría.

El código de Python 2 sangrado con una mezcla de tabulaciones y espacios debe convertirse para usar espacios exclusivamente.

Al invocar el intérprete de línea de comandos de Python 2 con la opción -t , emite advertencias sobre el código que mezcla ilegalmente tabulaciones y espacios. Cuando se usa -tt, estas advertencias se convierten en errores. ¡Estas opciones son muy recomendables!

Longitud máxima de la línea

Limite todas las líneas a un máximo de 79 caracteres.

Para bloques de texto largos fluidos con menos restricciones estructurales (cadenas de documentación o comentarios), la longitud de la línea debe limitarse a 72 caracteres.

Limitar el ancho de la ventana del editor requerido hace posible tener varios archivos abiertos uno al lado del otro y funciona bien cuando se utilizan herramientas de revisión de código que presentan las dos versiones en columnas adyacentes.

El ajuste predeterminado en la mayoría de las herramientas interrumpe la estructura visual del código, lo que lo hace más difícil de entender. Los límites se eligen para evitar que los editores se envuelvan con el ancho de ventana establecido en 80, incluso si la herramienta coloca un glifo de marcador en la columna final al ajustar las líneas. Es posible que algunas herramientas basadas en web no ofrezcan ningún ajuste de línea dinámico.

Algunos equipos prefieren encarecidamente una línea más larga. Para el código mantenido exclusiva o principalmente por un equipo que puede llegar a un acuerdo sobre este tema, está bien aumentar el límite de longitud de línea hasta 99 caracteres, siempre que los comentarios y las cadenas de documentos sigan envueltos en 72 caracteres.

La biblioteca estándar de Python es conservadora y requiere limitar las líneas a 79 caracteres (y cadenas de documentación / comentarios a 72).

La forma preferida de envolver líneas largas es utilizando la continuación de línea implícita de Python entre paréntesis, corchetes y llaves. Las líneas largas se pueden dividir en varias líneas ajustando las expresiones entre paréntesis. Estos deben usarse en lugar de usar una barra invertida para la continuación de la línea.

Las barras invertidas pueden ser apropiadas en ocasiones. Por ejemplo,
las declaraciones largas y múltiples `con` -no pueden usar continuación implícita, por lo que las barras diagonales inversas son aceptables:

```
con open ('/ ruta / a / algún / archivo / que / desea / a / leer') como
archivo_1, \

     abrir ('/ ruta / a / algún / archivo / siendo / escrito', 'w') como
archivo_2:

    file_2.write (file_1.read ())
```

(Consulte la discusión anterior sobre declaraciones if de varias líneas para obtener más información sobre la sangría de dichas declaraciones multilínea `con` -información).

Otro caso es la `aserción` declaraciones.

Asegúrese de sangrar la línea continua de manera adecuada.

¿Debe romperse una línea antes o después de un operador binario?

Durante décadas, el estilo recomendado fue romper con los operadores binarios. Pero esto puede dañar la legibilidad de dos maneras: los operadores tienden a dispersarse en diferentes columnas de la pantalla, y cada operador se aleja de su operando y pasa a la línea

anterior. Aquí, el ojo tiene que hacer un trabajo adicional para saber qué elementos se agregan y cuáles se restan:

```
# Equivocado:

# los operadores se sientan lejos de sus operandos

ingresos = (salario_bruto +

           intereses_impuestos +

           (dividendos - dividendos_calificados) -

           ira_deduction -

           Student_loan_interest)
```

Para resolver este problema de legibilidad, los matemáticos y sus editores siguen la convención opuesta. Donald Knuth explica la regla tradicional en su serie *Computers and Typesetting* : "Aunque las fórmulas dentro de un párrafo siempre se rompen después de las operaciones y relaciones binarias, las fórmulas mostradas siempre se rompen antes de las operaciones binarias" [3] .
Seguir la tradición de las matemáticas generalmente da como resultado un código más legible:

```
# Correcto:

# fácil de hacer coincidir operadores con operandos

ingresos = (salarios_brutos

           + interés_ imponible

           + (dividendos - dividendos_calificados)

           - ira_deduction

           - interés_préstamo_estudiante)
```

En el código Python, está permitido romper antes o después de un operador binario, siempre que la convención sea coherente localmente. Para el nuevo código, se sugiere el estilo de Knuth.

Líneas en blanco

Rodee las definiciones de clases y funciones de nivel superior con dos líneas en blanco.

Las definiciones de métodos dentro de una clase están rodeadas por una sola línea en blanco.

Se pueden usar (con moderación) líneas en blanco adicionales para separar grupos de funciones relacionadas. Se pueden omitir líneas en blanco entre un grupo de frases ingeniosas relacionadas (por ejemplo, un conjunto de implementaciones ficticias).

Use líneas en blanco en las funciones, con moderación, para indicar secciones lógicas.

Python acepta el carácter de alimentación de formulario control-L (es decir, ^ L) como espacio en blanco; Muchas herramientas tratan estos caracteres como separadores de página, por lo que puede usarlos para separar páginas de secciones relacionadas de su archivo. Tenga en cuenta que algunos editores y lectores de códigos basados en web pueden no reconocer control-L como un formulario de alimentación y mostrarán otro glifo en su lugar.

Codificación del archivo de origen

El código en la distribución central de Python siempre debe usar UTF-8 (o ASCII en Python 2).

Los archivos que usan ASCII (en Python 2) o UTF-8 (en Python 3) no deben tener una declaración de codificación.

En la biblioteca estándar, las codificaciones no predeterminadas deben usarse solo con fines de prueba o cuando un comentario o una cadena de documentos necesita mencionar un nombre de autor que contiene caracteres no ASCII; de lo contrario, el uso de escapes `\ x` , `\ u` , `\ U` o `\ N` es la forma preferida de incluir datos no ASCII en literales de cadena.

Para Python 3.0 y posteriores, se prescribe la siguiente política para la biblioteca estándar (ver PEP 3131): Todos los identificadores en la biblioteca estándar de Python DEBEN usar identificadores solo ASCII, y DEBEN usar palabras en inglés siempre que sea posible (en muchos casos, abreviaturas y se utilizan términos que no son en inglés). Además, los literales de cadena y los comentarios también deben estar en ASCII. Las únicas excepciones son (a) casos de prueba que prueban las características no ASCII y (b) nombres de autores. Los autores cuyos nombres no se basan en el alfabeto latino (latin-1, juego de caracteres ISO / IEC 8859-1) DEBEN proporcionar una transliteración de sus nombres en este juego de caracteres.

Se anima a los proyectos de código abierto con una audiencia global a adoptar una política similar.

Importaciones

- Las importaciones generalmente deben estar en líneas separadas:

```
# Correcto:

importar sistema operativo
```

```
importar sys

# Equivocado:

importar sys, os
```

Sin embargo, está bien decir esto:

```
# Correcto:

desde el subproceso de importación Popen, PIPE
```

- Las importaciones siempre se colocan en la parte superior del archivo, justo después de los comentarios y cadenas de documentación del módulo, y antes de las constantes y globales del módulo.

  Las importaciones deben agruparse en el siguiente orden:

  1. Importaciones de bibliotecas estándar.
  2. Importaciones de terceros relacionadas.
  3. Importaciones específicas de aplicaciones / bibliotecas locales.

  Debe poner una línea en blanco entre cada grupo de importaciones.

- Se recomiendan las importaciones absolutas, ya que generalmente son más legibles y tienden a comportarse mejor (o al menos dar mejores mensajes de error) si el sistema de importación está configurado incorrectamente (como cuando un directorio dentro de un paquete termina en `sys.path`):

```
importar mypkg.sibling

de mypkg import hermano

del ejemplo de importación mypkg.sibling
```

Sin embargo, las importaciones relativas explícitas son una alternativa aceptable a las importaciones absolutas, especialmente cuando se trata de diseños de paquetes complejos donde el uso de importaciones absolutas sería innecesariamente detallado:

```
desde . hermano de importación

de ejemplo de importación de .sibling
```

El código de biblioteca estándar debe evitar diseños de paquetes complejos y utilizar siempre importaciones absolutas.

Las importaciones relativas implícitas *nunca* deben usarse y se han eliminado en Python 3.

- Al importar una clase de un módulo que contiene una clase, generalmente está bien escribir esto:

```
- de myclass importar MyClass

- de foo.bar.yourclass importar YourClass
```

Si esta ortografía provoca conflictos de nombres locales, escríbalos explícitamente:

```
importar myclass

import foo.bar.yourclass
```

y use "myclass.MyClass" y "foo.bar.yourclass.YourClass".

- Deben evitarse las importaciones de comodines (`desde <module> import *`), ya que dejan poco claro qué nombres están presentes en el espacio de nombres, lo que confunde a los lectores y a muchas herramientas automatizadas. Existe un caso de uso defendible para una importación de comodines, que es volver a publicar una interfaz interna como parte de una API pública (por ejemplo, sobrescribir una implementación de Python pura de una interfaz con las definiciones de un módulo acelerador opcional y exactamente qué definiciones serán sobrescrito no se conoce de antemano).

Al volver a publicar nombres de esta manera, se siguen aplicando las siguientes pautas con respecto a las interfaces públicas e internas.

Nombres de Dunder a nivel de módulo

Los "errores" de nivel de módulo (es decir, los nombres con dos guiones bajos `iniciales` y dos finales) como `__todos__` , `__autor__` , `__versión__` , etc. deben colocarse después de la cadena de documentación del módulo pero antes de cualquier declaración de importación *excepto* `de las` importaciones de `__futuro__` . Python exige que las importaciones futuras deben aparecer en el módulo antes que cualquier otro código, excepto las cadenas de documentos:

```
"" "Este es el módulo de ejemplo.



Este módulo hace cosas.

"" "
```

```
desde __future__ import barry_as_FLUFL


__todos__ = ['a', 'b', 'c']

__version__ = '0.1'

__author__ = 'Cardenal Biggles'


importar sistema operativo

importar sys
```

Cotizaciones de cadena

En Python, las cadenas entre comillas simples y las cadenas entre comillas dobles son iguales. Este PEP no hace ninguna recomendación al respecto. Elija una regla y apéguese a ella. Sin embargo, cuando una cadena contiene caracteres de comillas simples o dobles, utilice la otra para evitar barras invertidas en la cadena. Mejora la legibilidad.

Para cadenas entre comillas triples, utilice siempre comillas dobles para ser coherente con la convención de cadenas de documentos en PEP 257 .

Espacio en blanco en expresiones y declaraciones

Mascota peeves

Evite los espacios en blanco extraños en las siguientes situaciones:

- Inmediatamente entre paréntesis, corchetes o llaves:

```
- # Correcto:
- spam (jamón [1], {huevos: 2})
- # Equivocado:
- spam (jamón [1], {huevos: 2})
```

- Entre una coma al final y un paréntesis cerrado siguiente:

```
# Correcto:

foo = (0,)

# Equivocado:

barra = (0,)
```

- Inmediatamente antes de una coma, punto y coma o dos puntos:

```
# Correcto:

si x == 4: imprimir x, y; x, y = y, x

# Equivocado:

si x == 4: imprimir x, y; x, y = y, x
```

- Sin embargo, en un segmento, los dos puntos actúan como un operador binario y deben tener cantidades iguales en ambos lados (tratándolo como el operador con la prioridad más baja). En un corte extendido, ambos dos puntos deben tener la misma cantidad de espacio aplicado. Excepción: cuando se omite un parámetro de sector, se omite el espacio:

```
# Correcto:

jamón [1: 9], jamón [1: 9: 3], jamón [: 9: 3], jamón [1 :: 3], jamón [1:
9:]

jamón [inferior: superior], jamón [inferior: superior:], jamón [inferior
:: escalón]

jamón [inferior + desplazamiento: superior + desplazamiento]

jamón [: superior_fn (x): paso_fn (x)], jamón [:: paso_fn (x)]

jamón [inferior + desplazamiento: superior + desplazamiento]

# Equivocado:

jamón [inferior + desplazamiento: superior + desplazamiento]

jamón [1: 9], jamón [1: 9], jamón [1: 9: 3]

jamón [inferior:: superior]
```

```
jamón [: superior]
```

- Inmediatamente antes del paréntesis abierto que inicia la lista de argumentos de una llamada a función:

```
# Correcto:
spam (1)
# Equivocado:
spam (1)
```

- Inmediatamente antes del paréntesis abierto que inicia una indexación o división:

```
# Correcto:
dct ['clave'] = lst [índice]
# Equivocado:
dct ['clave'] = lst [índice]
```

- Más de un espacio alrededor de un operador de asignación (u otro) para alinearlo con otro:

```
# Correcto:
x = 1
y = 2
variable_larga = 3
# Equivocado:
x = 1
y = 2
variable_larga = 3
```

Otras recomendaciones

- Evite dejar espacios en blanco en cualquier lugar. Debido a que generalmente es invisible, puede resultar confuso: por ejemplo, una barra invertida seguida de un espacio y una nueva línea no cuenta como un marcador de continuación de línea. Algunos editores no lo conservan y muchos proyectos (como el propio CPython) tienen ganchos de confirmación previa que lo rechazan.

- Siempre rodear estos operadores binarios con un solo espacio a cada lado: asignación (`=`), la asignación aumentada (`+ =` , `- =` , etc.), las comparaciones (`==` , `<` , `>` , `! =` , `<>` , `<=` , `> =` , `en` , `no en` , `es` , `no es`), booleanos (`y` , `o` , `no`).
- Si se utilizan operadores con diferentes prioridades, considere agregar espacios en blanco alrededor de los operadores con la (s) prioridad (es) más baja (s). Use su propio juicio; sin embargo, nunca use más de un espacio y siempre tenga la misma cantidad de espacios en blanco en ambos lados de un operador binario:

```
# Correcto:

yo = yo + 1

enviado + = 1

x = x * 2 - 1

hipot2 = x * x + y * y

c = (a + b) * (ab)

# Equivocado:

yo = yo + 1

enviado + = 1

x = x * 2 - 1

hipot2 = x * x + y * y

c = (a + b) * (a - b)
```

- Las anotaciones de funciones deben usar las reglas normales para dos puntos y siempre deben tener espacios alrededor de la flecha -> si está presente. (Consulte Anotaciones de funciones a continuación para obtener más información sobre las anotaciones de funciones):

```
# Correcto:

def munge (entrada: AnyStr): ...

def munge () -> PosInt: ...
```

```
# Equivocado:

def munge (entrada: AnyStr): ...

def munge () -> PosInt: ...
```

- No use espacios alrededor del signo = cuando se usa para indicar un argumento de palabra clave, o cuando se usa para indicar un valor predeterminado para un parámetro de función no *anotado* :

```
# Correcto:

def complejo (real, imag = 0.0):

    devolver magia (r = real, i = imag)

# Equivocado:

def complejo (real, imag = 0.0):

    devolver magia (r = real, i = imag)
```

Sin embargo, cuando combine una anotación de argumento con un valor predeterminado, use espacios alrededor del signo = :

```
# Correcto:

def munge (sep: AnyStr = None): ...

def munge (entrada: AnyStr, sep: AnyStr = Ninguno, límite = 1000): ...

# Equivocado:

def munge (entrada: AnyStr = Ninguno): ...

def munge (entrada: AnyStr, límite = 1000): ...
```

- Las declaraciones compuestas (declaraciones múltiples en la misma línea) generalmente se desaconsejan:

```
# Correcto:

si foo == 'blah':

    hacer_blah_thing ()
```

- has uno()
- do_two ()
- do_three ()

Mejor no:

```
# Equivocado:

si foo == 'blah': do_blah_thing ()

has uno(); do_two (); do_three ()
```

- Si bien a veces está bien poner un if / for / while con un cuerpo pequeño en la misma línea, nunca haga esto para declaraciones de múltiples cláusulas. ¡También evite doblar líneas tan largas!

Mejor no:

```
# Equivocado:

si foo == 'blah': do_blah_thing ()

para x en lst: total + = x

mientras t <10: t = retraso ()
```

Definitivamente no:

```
# Equivocado:

si foo == 'blah': do_blah_thing ()

más: do_non_blah_thing ()



intenta algo()

finalmente: limpieza ()



has uno(); do_two (); do_three (largo, argumento,
```

```
                                lista, como, esto)



si foo == 'blah': uno (); dos(); Tres()
```

Cuándo usar comas finales

Las comas finales suelen ser opcionales, excepto que son obligatorias al hacer una tupla de un elemento (y en Python 2 tienen semántica para la declaración de `impresión`). Para mayor claridad, se recomienda rodear este último entre paréntesis (técnicamente redundantes):

```
# Correcto:

ARCHIVOS = ('setup.cfg',)

# Equivocado:

ARCHIVOS = 'setup.cfg',
```

Cuando las comas finales son redundantes, a menudo son útiles cuando se utiliza un sistema de control de versiones, cuando se espera que una lista de valores, argumentos o elementos importados se amplíe con el tiempo. El patrón es poner cada valor (etc.) en una línea por sí mismo, siempre agregando una coma al final, y agregar el paréntesis / corchete / llave de cierre en la siguiente línea. Sin embargo, no tiene sentido tener una coma al final en la misma línea que el delimitador de cierre (excepto en el caso anterior de tuplas singleton):

```
# Correcto:

ARCHIVOS = [

    'setup.cfg',

    'tox.ini',

    ]

inicializar (ARCHIVOS,

           error = Verdadero,

           )

# Equivocado:

ARCHIVOS = ['setup.cfg', 'tox.ini',]
```

```
inicializar (ARCHIVOS, error = Verdadero,)
```

Comentarios

Los comentarios que contradicen el código son peores que ningún comentario. ¡Siempre tenga la prioridad de mantener los comentarios actualizados cuando cambie el código!

Los comentarios deben ser oraciones completas. La primera palabra debe escribirse en mayúscula, a menos que sea un identificador que comience con una letra minúscula (¡nunca altere el caso de los identificadores!).

Los comentarios en bloque generalmente consisten en uno o más párrafos construidos a partir de oraciones completas, y cada oración termina en un punto.

Debe utilizar dos espacios después de un período de finalización de una oración en los comentarios de varias oraciones, excepto después de la oración final.

Asegúrese de que sus comentarios sean claros y fácilmente comprensibles para otros hablantes del idioma en el que está escribiendo.

Codificadores de Python de países que no hablan inglés: escriba sus comentarios en inglés, a menos que esté 120% seguro de que el código nunca será leído por personas que no hablen su idioma.

Bloquear comentarios

Los comentarios de bloque generalmente se aplican a algunos (o todos) códigos que los siguen, y están sangrados al mismo nivel que ese código. Cada línea de un comentario de bloque comienza con un `#` y un solo espacio (a menos que sea texto sangrado dentro del comentario).

Los párrafos dentro de un comentario de bloque están separados por una línea que contiene un solo `#` .

Comentarios en línea

Utilice los comentarios en línea con moderación.

Un comentario en línea es un comentario en la misma línea que una declaración. Los comentarios en línea deben estar separados por al menos dos espacios de la declaración. Deben comenzar con un # y un solo espacio.

Los comentarios en línea son innecesarios y de hecho distraen si dicen lo obvio. No hagas esto:

```
x = x + 1 # Incremento x
```

Pero a veces, esto es útil:

```
x = x + 1 # Compensa por borde
```

Cadenas de documentación

Las convenciones para escribir buenas cadenas de documentación (también conocidas como "cadenas de documentación") están inmortalizadas en PEP 257 .

- Escriba cadenas de documentación para todos los módulos, funciones, clases y métodos públicos. Las cadenas de documentos no son necesarias para los métodos no públicos, pero debe tener un comentario que describa lo que hace el método. Este comentario debería aparecer después de la línea `def` .

- PEP 257 describe buenas convenciones de cadenas de documentos. Tenga en cuenta que lo más importante es que el `"" "` que termina una cadena de documentos de varias líneas debe estar en una línea por sí mismo:

```
- "" "Devolver un foobang
-
- El plotz opcional dice frobnicate el bizbaz primero.
- "" "
```

- Para cadenas de documentos de una línea, mantenga el `"" "de` cierre en la misma línea:

```
- "" "Devuelve un ex-loro." ""
```

Convenciones de nombres

Las convenciones de nomenclatura de la biblioteca de Python son un poco desordenadas, por lo que nunca obtendremos esto completamente consistente; sin embargo, aquí están los estándares de nomenclatura recomendados actualmente. Los nuevos módulos y paquetes (incluidos los marcos de terceros) deben escribirse de acuerdo con estos estándares, pero cuando una biblioteca existente tiene un estilo diferente, se prefiere la coherencia interna.

Principio primordial

Los nombres que son visibles para el usuario como partes públicas de la API deben seguir convenciones que reflejen el uso en lugar de la implementación.

Descriptivo: Estilos de nombres

Hay muchos estilos de nombres diferentes. Es útil poder reconocer qué estilo de nomenclatura se está utilizando, independientemente de para qué se utilizan.

Los siguientes estilos de nomenclatura se distinguen comúnmente:

- `b` (una sola letra minúscula)
- `B` (letra mayúscula única)
- `minúscula`
- `minúsculas_con_puntuaciones`
- `MAYÚSCULAS`
- `UPPER_CASE_WITH_UNDERSCORES`
- Palabras en `mayúsculas` (o CapWords, o CamelCase - así llamado por el aspecto irregular de sus letras [4]). Esto también se conoce a veces como StudlyCaps.

  Nota: Cuando utilice siglas en CapWords, escriba en mayúscula todas las letras de las siglas. Por tanto, HTTPServerError es mejor que HttpServerError.

- `MixedCase` (difiere de CapitalizedWords por el carácter inicial en minúscula)
- `Capitalized_Words_With_Underscores` (¡feo!)

También existe el estilo de usar un prefijo único corto para agrupar nombres relacionados. Esto no se usa mucho en Python, pero se menciona para completarlo. Por ejemplo, la función `os.stat ()` devuelve una tupla cuyos elementos tradicionalmente tienen nombres como `st_mode` , `st_size` , `st_mtime` , etc. (Esto se hace para enfatizar la correspondencia con los campos de la estructura de llamada del sistema POSIX, lo que ayuda a los programadores familiarizados con eso).

La biblioteca X11 utiliza una X inicial para todas sus funciones públicas. En Python, este estilo generalmente se considera innecesario porque los nombres de atributo y método tienen como prefijo un objeto, y los nombres de función tienen como prefijo un nombre de módulo.

Además, se reconocen las siguientes formas especiales que utilizan guiones bajos al principio o al final (generalmente se pueden combinar con cualquier convención de casos):

- `_single_leading_underscore` : `indicador` débil de "uso interno". Por ejemplo, `de M import *` no importa objetos cuyos nombres comiencen con un guión bajo.
- `single_trailing_underscore_` : utilizado por convención para evitar conflictos con la palabra clave de Python, por ejemplo

```
tkinter.Toplevel (maestro, clase _ = 'ClassName')
```

- `__double_leading_underscore` : al nombrar un atributo de clase, invoca el cambio de nombre (dentro de la clase FooBar, `__boo se` convierte en `_FooBar__boo` ; ver más abajo).

- `__double_leading_and_trailing_underscore__` : objetos o atributos "mágicos" que viven en espacios de nombres controlados por el usuario. Por `ejemplo` , `__init__` , `__import__` o `__file__` . Nunca invente tales nombres; utilícelos únicamente según lo documentado.

Prescriptivo: convenciones de nomenclatura

Nombres para evitar

Nunca use los caracteres 'l' (letra minúscula el), 'O' (letra mayúscula oh) o 'I' (letra mayúscula ojo) como nombres de variable de un solo carácter.

En algunas fuentes, estos caracteres son indistinguibles de los números uno y cero. Cuando tenga la tentación de usar 'l', use 'L' en su lugar.

Compatibilidad ASCII

Los identificadores utilizados en la biblioteca estándar deben ser compatibles con ASCII como se describe en la sección de políticas de PEP 3131 .

Nombres de módulos y paquetes

Los módulos deben tener nombres cortos en minúsculas. Se pueden usar guiones bajos en el nombre del módulo si mejora la legibilidad. Los paquetes de Python también deben tener nombres cortos en minúsculas, aunque se desaconseja el uso de guiones bajos.

Cuando un módulo de extensión escrito en C o C ++ tiene un módulo Python adjunto que proporciona una interfaz de nivel superior (por ejemplo, más orientada a objetos), el módulo C / C ++ tiene un subrayado `inicial` (por ejemplo, `_socket`).

Nombres de clases

Los nombres de las clases normalmente deben usar la convención de CapWords.

La convención de nomenclatura para funciones se puede usar en cambio en los casos en que la interfaz está documentada y se usa principalmente como un invocable.

Tenga en cuenta que existe una convención separada para los nombres incorporados: la mayoría de los nombres incorporados son palabras individuales (o dos palabras se ejecutan juntas), y la convención de CapWords se usa solo para nombres de excepción y constantes incorporadas.

Escriba nombres de variables

Los nombres de las variables de tipo introducidas en PEP 484 normalmente deben usar CapWords prefiriendo nombres cortos: `T` , `AnyStr` , `Num` . Se recomienda agregar los sufijos `_co` o `_contra` a las variables utilizadas para declarar el comportamiento covariante o contravariante correspondientemente:

```
escribiendo import TypeVar



VT_co = TypeVar ('VT_co', covariante = Verdadero)

KT_contra = TypeVar ('KT_contra', contravariant = True)
```

Nombres de excepción

Debido a que las excepciones deben ser clases, aquí se aplica la convención de nomenclatura de clases. Sin embargo, debe utilizar el sufijo "Error" en los nombres de sus excepciones (si la excepción en realidad es un error).

Nombres de variables globales

(Esperemos que estas variables estén diseñadas para su uso dentro de un solo módulo). Las convenciones son aproximadamente las mismas que las de las funciones.

Los módulos que están diseñados para usarse a través `de M import *` deben usar el mecanismo `__todos__` para evitar la exportación de globales, o usar la convención anterior de prefijar dichos globales con un guión bajo (lo que puede hacer para indicar que estos globales son "módulo no público ").

Nombres de funciones y variables

Los nombres de las funciones deben estar en minúsculas, con palabras separadas por guiones bajos según sea necesario para mejorar la legibilidad.

Los nombres de las variables siguen la misma convención que los nombres de las funciones.

MixedCase solo se permite en contextos donde ese ya es el estilo predominante (por ejemplo, threading.py), para mantener la compatibilidad con versiones anteriores.

Argumentos de función y método

Utilice siempre `self` para el primer argumento de los métodos de instancia.

Utilice siempre `cls` para el primer argumento de los métodos de clase.

Si el nombre de un argumento de función choca con una palabra clave reservada, generalmente es mejor agregar un guión bajo al final en lugar de usar una abreviatura o una

alteración ortográfica. Por `lo` tanto, `class_` es mejor que `clss` . (Quizás sea mejor evitar tales choques usando un sinónimo).

Nombres de métodos y variables de instancia

Utilice las reglas de denominación de funciones: minúsculas con palabras separadas por guiones bajos según sea necesario para mejorar la legibilidad.

Utilice un guión bajo inicial solo para métodos no públicos y variables de instancia.

Para evitar conflictos de nombres con subclases, use dos guiones bajos iniciales para invocar las reglas de modificación de nombres de Python.

Python modifica estos nombres con el nombre de la clase: si la clase Foo tiene un atributo llamado `__a` , `Foo` no puede acceder a él `.__ a` . (Un usuario insistente aún podría obtener acceso llamando a `Foo._Foo__a` .) Generalmente, los guiones bajos `iniciales` dobles deben usarse solo para evitar conflictos de nombres con atributos en clases diseñadas para ser subclasificadas.

Nota: existe cierta controversia sobre el uso de __names (ver más abajo).

Constantes

Las constantes generalmente se definen a nivel de módulo y se escriben en letras mayúsculas con guiones bajos que separan las palabras. Los ejemplos incluyen `MAX_OVERFLOW` y `TOTAL` .

Diseñar para la herencia

Decida siempre si los métodos de una clase y las variables de instancia (colectivamente: "atributos") deben ser públicos o no públicos. En caso de duda, elija no público; es más fácil hacerlo público más tarde que convertir un atributo público en no público.

Los atributos públicos son aquellos que espera que usen clientes de su clase no relacionados, con su compromiso de evitar cambios incompatibles con versiones anteriores. Los atributos no públicos son aquellos que no están destinados a ser utilizados por terceros; no ofrece garantías de que los atributos no públicos no cambiarán o incluso se eliminarán.

No usamos el término "privado" aquí, ya que ningún atributo es realmente privado en Python (sin una cantidad de trabajo generalmente innecesaria).

Otra categoría de atributos son los que forman parte de la "subclase API" (a menudo denominada "protegida" en otros idiomas). Algunas clases están diseñadas para heredarse, ya sea para ampliar o modificar aspectos del comportamiento de la clase. Al diseñar una clase de este tipo, tenga cuidado de tomar decisiones explícitas sobre qué atributos son públicos, cuáles son parte de la API de subclase y cuáles realmente solo deben ser utilizados por su clase base.

Con esto en mente, aquí están las pautas Pythonic:

- Los atributos públicos no deben tener guiones bajos iniciales.

- Si el nombre de su atributo público choca con una palabra clave reservada, agregue un guión bajo al final de su nombre de atributo. Esto es preferible a una abreviatura o una ortografía corrupta. (Sin embargo, a pesar de esta regla, 'cls' es la ortografía preferida para cualquier variable o argumento que se sepa que es una clase, especialmente el primer argumento de un método de clase).

  Nota 1: Consulte la recomendación anterior sobre el nombre del argumento para conocer los métodos de clase.

- Para atributos de datos públicos simples, es mejor exponer solo el nombre del atributo, sin métodos complicados de acceso / mutador. Tenga en cuenta que Python proporciona un camino fácil para futuras mejoras, en caso de que descubra que un atributo de datos simple necesita desarrollar un comportamiento funcional. En ese caso, utilice propiedades para ocultar la implementación funcional detrás de una sintaxis de acceso a atributos de datos simple.

  Nota 1: Las propiedades solo funcionan en clases de estilo nuevo.

  Nota 2: Intente mantener libre de efectos secundarios de comportamiento funcional, aunque los efectos secundarios como el almacenamiento en caché generalmente están bien.

  Nota 3: Evite el uso de propiedades para operaciones computacionalmente costosas; la notación de atributo hace que la persona que llama crea que el acceso es (relativamente) barato.

- Si su clase está destinada a ser subclasificada y tiene atributos que no desea que utilicen las subclases, considere nombrarlos con guiones bajos iniciales dobles y sin guiones bajos finales. Esto invoca el algoritmo de manipulación de nombres de Python, donde el nombre de la clase se transforma en el nombre del atributo. Esto ayuda a evitar colisiones de nombres de atributos en caso de que las subclases contengan inadvertidamente atributos con el mismo nombre.

  Nota 1: Tenga en cuenta que solo se usa el nombre de clase simple en el nombre mutilado, por lo que si una subclase elige el mismo nombre de clase y nombre de atributo, aún puede obtener colisiones de nombres.

  Nota 2: La alteración de nombres puede hacer que ciertos usos, como la depuración y `__getattr __ ()` , sean menos convenientes. Sin embargo, el algoritmo de manipulación de nombres está bien documentado y es fácil de realizar manualmente.

  Nota 3: No a todo el mundo le gusta la manipulación de nombres. Intente equilibrar la necesidad de evitar conflictos de nombres accidentales con el uso potencial por parte de personas avanzadas.

Interfaces públicas e internas

Las garantías de compatibilidad con versiones anteriores se aplican solo a las interfaces públicas. En consecuencia, es importante que los usuarios puedan distinguir claramente entre interfaces públicas e internas.

Las interfaces documentadas se consideran públicas, a menos que la documentación las declare explícitamente como provisionales o interfaces internas exentas de las habituales garantías de retrocompatibilidad. Se debe suponer que todas las interfaces no documentadas son internas.

Para soportar mejor la introspección, los módulos deben declarar explícitamente los nombres en su API pública usando el atributo `__all__` . Establecer `__all__` en una lista vacía indica que el módulo no tiene API pública.

Incluso con `__todos__` configurados apropiadamente, las interfaces internas (paquetes, módulos, clases, funciones, atributos u otros nombres) aún deben tener un prefijo con un solo subrayado inicial.

Una interfaz también se considera interna si cualquier espacio de nombres que contenga (paquete, módulo o clase) se considera interno.

Los nombres importados siempre deben considerarse un detalle de implementación. Otros módulos no deben depender del acceso indirecto a dichos nombres importados a menos que sean una parte explícitamente documentada de la API del módulo contenedor, como `os.path` o el módulo `__init__ de` un paquete que exponga la funcionalidad de los submódulos.

Recomendaciones de programación

- El código debe estar escrito de una manera que no perjudique a otras implementaciones de Python (PyPy, Jython, IronPython, Cython, Psyco, etc.).

  Por ejemplo, no confíe en la implementación eficiente de CPython de la concatenación de cadenas en el lugar para declaraciones en la forma `a + = b` o `a = a + b` . Esta optimización es frágil incluso en CPython (solo funciona para algunos tipos) y no está presente en absoluto en implementaciones que no usan refcounting. En las partes sensibles al rendimiento de la biblioteca, se debe usar la forma " .join () en su lugar. Esto garantizará que la concatenación se produzca en tiempo lineal en varias implementaciones.
- Las comparaciones con singletons como None siempre deben hacerse con `es` o `no es` , nunca con operadores de igualdad.

  Además, tenga cuidado con escribir `si x` cuando realmente quiere decir `si x no es Ninguno` , por ejemplo, cuando se prueba si una variable o argumento que por defecto es Ninguno se estableció en algún otro valor. El otro valor podría tener un tipo (como un contenedor) que podría ser falso en un contexto booleano.

- El uso `no es` operador en lugar de `no ... es` . Si bien ambas expresiones son funcionalmente idénticas, la primera es más legible y preferida:

```
# Correcto:

si foo no es None:

# Equivocado:

si no foo es Ninguno:
```

- Al implementar operaciones de orden con comparaciones enriquecidas, es mejor implementar las seis operaciones (`__eq__` , `__ne__` , `__lt__` , `__le__` , `__gt__` , `__ge__`) en lugar de depender de otro código para realizar solo una comparación en particular.

  Para minimizar el esfuerzo involucrado, el decorador `functools.total_ordering ()` proporciona una herramienta para generar métodos de comparación faltantes.

  PEP 207 indica que las reglas de reflexividad *se* asumidas por Python. Por lo tanto, el intérprete puede intercambiar `y> x` con `x <y` , `y> = x` con `x <= y` , y puede intercambiar los argumentos de `x == y` y `x! = Y` . Las operaciones `sort ()` y `min ()` están garantizadas para usar el operador `<` y la función `max ()` usa el operador `>` . Sin embargo, es mejor implementar las seis operaciones para que no surja confusión en otros contextos.
- Utilice siempre una declaración def en lugar de una declaración de asignación que vincule una expresión lambda directamente a un identificador:

```
# Correcto:

def f (x): devuelve 2 * x

# Equivocado:

f = lambda x: 2 * x
```

La primera forma significa que el nombre del objeto de función resultante es específicamente 'f' en lugar del genérico '<lambda>'. Esto es más útil para trazas y representaciones de cadenas en general. El uso de la declaración de asignación elimina el único beneficio que una expresión lambda puede ofrecer sobre una declaración def explícita (es decir, que se puede incrustar dentro de una expresión más grande)

- Derive excepciones de `Exception en` lugar de `BaseException` . La herencia directa de `BaseException` está reservada para excepciones en las que capturarlas es casi siempre lo incorrecto.

  Diseñe jerarquías de excepciones basadas en las distinciones que probablemente necesite el código que *detecta* las excepciones, en lugar de las ubicaciones donde se generan las excepciones. Trate de responder a la pregunta "¿Qué salió mal?" programáticamente, en lugar de solo indicar que "Ocurrió un problema"

(consulte PEP 3151 para ver un ejemplo de cómo se aprende esta lección para la jerarquía de excepciones incorporada)
Las convenciones de nomenclatura de clases se aplican aquí, aunque debe agregar el sufijo "Error" a sus clases de excepción si la excepción es un error. Las excepciones sin errores que se utilizan para el control de flujo no local u otras formas de señalización no necesitan un sufijo especial.

- Utilice el encadenamiento de excepciones de forma adecuada. En Python 3, "subir X desde Y" debe usarse para indicar un reemplazo explícito sin perder el rastreo original.

  Al reemplazar deliberadamente una excepción interna (usando "subir X" en Python 2 o "subir X desde Ninguno" en Python 3.3+), asegúrese de que los detalles relevantes se transfieran a la nueva excepción (como preservar el nombre del atributo al convertir KeyError a AttributeError o incrustando el texto de la excepción original en el nuevo mensaje de excepción).

- Cuando genere una excepción en Python 2, use `raise` ValueError ('mensaje') en lugar de la forma anterior `raise ValueError, 'mensaje'`.
  La última forma no es la sintaxis legal de Python 3.

  La forma de uso de par también significa que cuando los argumentos de excepción son largos o incluyen formato de cadena, no es necesario usar caracteres de continuación de línea gracias a los paréntesis que los contienen.

- Cuando detecte excepciones, mencione excepciones específicas siempre que sea posible en lugar de usar una cláusula `excepto` :

```
intentar:
    importar módulo_específico_de_plataforma
excepto ImportError:
    platform_specific_module = Ninguno
```

Una simple cláusula `except:` capturará las excepciones SystemSalir y KeyboardInterrupt, dificultando la interrupción de un programa con Control-C, y puede ocultar otros problemas. Si desea capturar todas las excepciones que señalan errores de programa, use `except Exception:` (bare except es equivalente a `except BaseException:`).

Una buena regla general es limitar el uso de cláusulas "excepto" desnudas a dos casos:

1. Si el manejador de excepciones imprimirá o registrará el rastreo; al menos el usuario será consciente de que se ha producido un error.

2. Si el código necesita hacer algún trabajo de limpieza, pero luego permite que la excepción se propague hacia arriba con `raise` . probar ... finalmente puede ser una mejor manera de manejar este caso.

- Al vincular excepciones detectadas a un nombre, prefiera la sintaxis de vinculación de nombre explícita agregada en Python 2.6:

```
intentar:
    procesar datos()
excepto Exception as exc:
    elevar DataProcessingFailedError (str (exc))
```

Esta es la única sintaxis compatible con Python 3 y evita los problemas de ambigüedad asociados con la sintaxis antigua basada en comas.

- Al detectar errores del sistema operativo, prefiera la jerarquía de excepción explícita introducida en Python 3.3 sobre la introspección de los valores de `errno` .
- Además, para todas las cláusulas try / except, limite la cláusula `try` a la cantidad mínima absoluta de código necesaria. Nuevamente, esto evita enmascarar errores:

```
# Correcto:
intentar:
    valor = colección [clave]
excepto KeyError:
    return key_not_found (clave)
demás:
    return handle_value (valor)
# Equivocado:
intentar:
    # ¡Demasiado amplia!
    return handle_value (colección [clave])
excepto KeyError:
```

```
    # También detectará KeyError generado por handle_value ()
    return key_not_found (clave)
```

- Cuando un recurso es local para una sección particular de código, use una instrucción `with` para asegurarse de que se limpie de manera rápida y confiable después de su uso. También es aceptable una declaración de intento / finalización.

- Los administradores de contexto deben invocarse a través de funciones o métodos separados siempre que hagan algo más que adquirir y liberar recursos:

```
# Correcto:
con conn.begin_transaction ():
    do_stuff_in_transaction (conexión)
# Equivocado:
con conexión:
    do_stuff_in_transaction (conexión)
```

El último ejemplo no proporciona ninguna información que indique que los métodos `__enter__` y `__exit__` están haciendo algo más que cerrar la conexión después de una transacción. Ser explícito es importante en este caso.

- Sea coherente en las declaraciones de devolución. O todas las declaraciones de retorno en una función deben devolver una expresión, o ninguna de ellas debe hacerlo. Si alguna declaración de devolución devuelve una expresión, cualquier declaración de devolución en la que no se devuelva ningún valor debe indicarlo explícitamente como `return None` , y una declaración de devolución explícita debe estar presente al final de la función (si es accesible):

```
# Correcto:

def foo (x):
    si x> = 0:
        devuelve math.sqrt (x)
    demás:
        regresar Ninguno
```

```
def bar (x):
    si x <0:
        regresar Ninguno
    devuelve math.sqrt (x)
# Equivocado:

def foo (x):
    si x> = 0:
        devuelve math.sqrt (x)

def bar (x):
    si x <0:
        regreso
    devuelve math.sqrt (x)
```

- Utilice métodos de cadena en lugar del módulo de cadena.

  Los métodos de cadena son siempre mucho más rápidos y comparten la misma API con cadenas Unicode. Anule esta regla si se requiere compatibilidad con versiones anteriores de Pythons anteriores a 2.0.

- Use " .startswith () y " .endswith () en lugar de cortar cadenas para buscar prefijos o sufijos.
  startswith () y endswith () son más limpios y menos propensos a errores:

```
# Correcto:
si foo.startswith ('bar'):
# Equivocado:
si foo [: 3] == 'bar':
```

- Las comparaciones de tipos de objetos siempre deben usar isinstance () en lugar de comparar tipos directamente:

```
# Correcto:
si es instancia (obj, int):
# Equivocado:
si el tipo (obj) es el tipo (1):
```

Cuando compruebe si un objeto es una cadena, tenga en cuenta que también podría ser una cadena Unicode. En Python 2, str y unicode tienen una clase base común, basetring, por lo que puede hacer:

```
si es instancia (obj, cadena base):
```

Tenga en cuenta que en Python 3, `unicode` y la `cadena base` ya no existen (solo hay `str`) y un objeto de bytes ya no es un tipo de cadena (es una secuencia de enteros en su lugar).

- Para las secuencias (cadenas, listas, tuplas), utilice el hecho de que las secuencias vacías son falsas:

```
# Correcto:
si no es seq:
si seq:
# Equivocado:
si len (seq):
si no len (seq):
```

- No escriba cadenas literales que dependan de espacios en blanco finales significativos. Estos espacios en blanco finales son visualmente indistinguibles y algunos editores (o más recientemente, reindent.py) los recortarán.
- No compare los valores booleanos con Verdadero o Falso usando == :

```
# Correcto:
```

```
si saludo:

# Equivocado:

si saludo == Verdadero:
```

Peor:

```
# Equivocado:

si el saludo es verdadero:
```

- Se desaconseja el uso de las sentencias de control de flujo `return` / `break` / `continue` dentro del conjunto final de un intento ... finalmente , cuando la declaración de control de flujo saltaría fuera del conjunto final. Esto se debe a que tales declaraciones cancelarán implícitamente cualquier excepción activa que se propague a través de la suite finalmente:

```
# Equivocado:

def foo ():

    intentar:

        1/0

    por fin:

        volver 42
```

Anotaciones de funciones

Con la aceptación de PEP 484 , las reglas de estilo para las anotaciones de funciones están cambiando.

- Para ser compatibles con versiones posteriores, las anotaciones de funciones en el código Python 3 deben usar preferiblemente la sintaxis PEP 484 . (Hay algunas recomendaciones de formato para las anotaciones en la sección anterior).

- Ya no se fomenta la experimentación con estilos de anotaciones que se recomendaba anteriormente en este PEP.

- Sin embargo, fuera de stdlib, ahora se recomiendan los experimentos dentro de las reglas de PEP 484 . Por ejemplo, marcar una gran biblioteca o aplicación de terceros con anotaciones de tipo de estilo PEP 484 , revisar lo fácil que fue agregar esas anotaciones y observar si su presencia aumenta la comprensibilidad del código.

- La biblioteca estándar de Python debería ser conservadora al adoptar tales anotaciones, pero su uso está permitido para código nuevo y para grandes refactorizaciones.

- Para el código que quiera hacer un uso diferente de las anotaciones de funciones se recomienda poner un comentario del formulario:

```
# tipo: ignorar
```

cerca de la parte superior del archivo; esto le dice al verificador de tipos que ignore todas las anotaciones. (Se pueden encontrar formas más detalladas de deshabilitar las quejas de los verificadores de tipo en PEP 484).

- Al igual que los linters, los comprobadores de tipos son herramientas opcionales e independientes. Los intérpretes de Python de forma predeterminada no deberían emitir ningún mensaje debido a la verificación de tipos y no deberían alterar su comportamiento en función de las anotaciones.

- Los usuarios que no quieran utilizar verificadores de tipo pueden ignorarlos. Sin embargo, se espera que los usuarios de paquetes de bibliotecas de terceros quieran ejecutar verificadores de tipo sobre esos paquetes. Para este propósito, PEP 484 recomienda el uso de archivos stub: archivos .pyi que el verificador de tipos lee con preferencia a los archivos .py correspondientes. Los archivos stub se pueden distribuir con una biblioteca o por separado (con el permiso del autor de la biblioteca) a través del repositorio mecanografiado [5] .

- Para el código que debe ser compatible con versiones anteriores, se pueden agregar anotaciones de tipo en forma de comentarios. Consulte la sección correspondiente de PEP 484 [6] .

Anotaciones variables

PEP 526 introdujo anotaciones variables. Las recomendaciones de estilo para ellos son similares a las de las anotaciones de funciones descritas anteriormente:

- Las anotaciones para variables de nivel de módulo, variables de clase e instancia y variables locales deben tener un solo espacio después de los dos puntos.

- No debe haber espacio antes de los dos puntos.

- Si una tarea tiene un lado derecho, entonces el signo de igualdad debe tener exactamente un espacio en ambos lados:

```
# Correcto:


código: int
```

```
punto de clase:
    coords: Tuple [int, int]
    etiqueta: str = '<desconocido>'
# Equivocado:

código: int # Sin espacio después de dos puntos
código: int # Espacio antes de los dos puntos

prueba de clase:
    resultado: int = 0 # Sin espacios alrededor del signo de igualdad
```

- Aunque el PEP 526 se acepta para Python 3.6, la sintaxis de anotación de variable es la sintaxis preferida para los archivos stub en todas las versiones de Python (consulte PEP 484 para obtener más detalles).

Notas al pie

[7] *La sangría francesa* es un estilo de tipografía en el que todas las líneas de un párrafo tienen sangría excepto la primera línea. En el contexto de Python, el término se utiliza para describir un estilo en el que el paréntesis de apertura de una declaración entre paréntesis es el último carácter de la línea que no es un espacio en blanco, y las líneas siguientes se sangran hasta el paréntesis de cierre.

Referencias

[1] PEP 7 , Guía de estilo para código C, van Rossum

[2] Guía de estilo de GNU Mailman de Barry http://barry.warsaw.us/software/STYLEGUIDE.txt

[3] *The TeXBook de* Donald Knuth , páginas 195 y 196.

[4] http://www.wikipedia.com/wiki/CamelCase

[5] Repositorio de Typehed https://github.com/python/typeshed

[6]	Sintaxis sugerida para Python 2.7 y código transversal https://www.python.org/dev/peps/pep-0484/#suggested-syntax-for-python-2-7-and-straddling-code

Derechos de autor